NumPy Fundamentals for Machine Learning

> by: Saeed Mohagheghi + AI

NumPy (Numerical Python) is the foundational package for numerical computing in Python. It provides an efficient way to store and manipulate large arrays of numerical data, which is crucial for machine learning tasks like data preprocessing, model training, and evaluation.

Let's dive into the core concepts with practical examples.

1. Importing NumPy and Creating Arrays

The standard convention is to import NumPy as `np`. The primary data structure in NumPy is the `ndarray` (N-dimensional array).

```
import numpy as np

# Creating a 1-dimensional array (vector) from a Python list
a = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
print("Array 'a':", a)
print("Type of 'a':", type(a))

# Creating a 2-dimensional array (matrix) from a list of lists
b = np.array([[10, 20, 30], [40, 50, 60]])
print("\nArray 'b':\n", b)
```

2. Array Attributes: `shape`, `dtype`, `astype`

NumPy arrays have several useful attributes:

- `shape`: A tuple indicating the size of the array in each dimension.
- `dtype`: The data type of the elements in the array (e.g., `int32`, `float64`). All elements in a NumPy array must have the same `dtype`.
- `astype`: A method to explicitly change the data type of an array.

```
# Using array 'a' from above: np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
print("\n--- Array Attributes ---")
print("Shape of 'a':", a.shape)
print("Data type of 'a':", a.dtype)

# Using array 'b' from above: np.array([[10, 20, 30], [40, 50, 60]])
print("Shape of 'b':", b.shape)
print("Data type of 'b':", b.dtype)

# Changing data type using astype()
c = a.astype(np.float64)
print("\nArray 'c' (a as float64):", c)
print("Data type of 'c':", c.dtype)

# Example: Converting a float array to integer (truncates decimal part)
d = np.array([1.1, 2.9, 3.5])
e = d.astype(np.int32)
print("\nArray 'd':", d)
print("Array 'e' (d as int32):", e)
```

3. Array Creation Functions

NumPy provides convenient functions to create arrays with specific patterns.

- `np.linspace(start, stop, num)`: Returns `num` evenly spaced samples, calculated over the interval `[start, stop]`.
- `np.arange(start, stop, step)`: Returns evenly spaced values within a given interval. Similar to Python's `range()`.
- `np.zeros(shape)`: Creates an array filled with zeros.
- `np.ones(shape)`: Creates an array filled with ones.

```
print("\n--- Array Creation Functions ---")

# linspace: 5 evenly spaced numbers between 0 and 10
lin_array = np.linspace(0, 10, 5)
print("np.linspace(0, 10, 5):", lin_array)

# arange: numbers from 0 up to (but not including) 10, with a step of 2
arr_array = np.arange(0, 10, 2)
print("np.arange(0, 10, 2):", arr_array)

# zeros: a 3x4 array of zeros
zeros_array = np.zeros((3, 4))
print("\nnp.zeros((3, 4)):\n", zeros_array)

# ones: a 2x3 array of ones
ones_array = np.ones((2, 3))
print("\nnp.ones((2, 3)):\n", ones_array)
```

4. Reshaping Arrays

The `reshape()` method allows you to change the shape of an array without changing its data. A special value `-1` can be used to automatically calculate the dimension.

- `reshape((n, -1))`: Reshapes the array into `n` rows, with the number of columns automatically determined.
- `reshape((-1, n))`: Reshapes the array into `n` columns, with the number of rows automatically determined.

```
# Let's use our 'a' array again: np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
print("\n--- Reshaping Arrays ---")
print("Original array 'a':", a)
print("Original shape of 'a':", a.shape)

# Reshape 'a' into 2 rows, automatically determine columns
reshaped_a_2_rows = a.reshape((2, -1))
print("\nReshaped 'a' to (2, -1):\n", reshaped_a_2_rows)
print("Shape:", reshaped_a_2_rows.shape)

# Reshape 'a' into 5 columns, automatically determine rows
reshaped_a_5_cols = a.reshape((-1, 5))
print("\nReshaped 'a' to (-1, 5):\n", reshaped_a_5_cols)
print("Shape:", reshaped_a_5_cols.shape)

# Reshape 'a' into a 3D array (2x5 elements, 10 total)
# This example requires 2*5 = 10 elements, which matches 'a'
reshaped_a_3d = a.reshape((2, 5, 1))
print("\nReshaped 'a' to (2, 5, 1):\n", reshaped_a_3d)
print("Shape:", reshaped_a_3d.shape)
```

5. Indexing and Slicing

Accessing elements or subsets of arrays is fundamental.

- **Basic Slicing**: Similar to Python lists, `[start:end:step]`.
- **Fancy Indexing**: Using an array of integers to select specific elements.
- **Boolean Indexing**: Using a boolean array (of the same shape) to select elements where the boolean array is `True`.
- `axis`: Many NumPy operations can be performed along a specific axis (dimension).
  - `axis=0` refers to operations along rows (column-wise).
  - `axis=1` refers to operations along columns (row-wise).

```
# Let's create a new array for indexing examples
data = np.arange(1, 26).reshape(5, 5)
print("\n--- Indexing and Slicing ---")
print("Original 'data' array (5x5):\n", data)

# Basic Slicing: Select first 3 rows and all columns
slice_1 = data[:3, :]
print("\nSlice [0:3, :]:\n", slice_1)

# Basic Slicing: Select rows from index 1 to 3 (exclusive) and columns from index 2 onwards
slice_2 = data[1:4, 2:]
print("\nSlice [1:4, 2:]:\n", slice_2)

# Fancy Indexing: Select specific rows (1, 3, 0)
fancy_index_rows = data[[1, 3, 0], :]
print("\nFancy Indexing (rows 1, 3, 0):\n", fancy_index_rows)

# Fancy Indexing: Select specific elements at (row, col) pairs
# (0,0), (1,2), (2,4)
fancy_index_elements = data[[0, 1, 2], [0, 2, 4]]
print("\nFancy Indexing (elements at (0,0), (1,2), (2,4)):\n", fancy_index_elements)

# Boolean Indexing: Select elements greater than 15
bool_mask = data > 15
print("\nBoolean mask (data > 15):\n", bool_mask)
bool_indexed_data = data[bool_mask]
print("Elements greater than 15:\n", bool_indexed_data)

# Boolean Indexing with logical operators: Elements between 5 and 15 (inclusive)
# Note: Use '&' for element-wise logical AND, '|' for element-wise logical OR
bool_mask_combined = (data >= 5) & (data <= 15)
print("\nBoolean mask (data >= 5 & data <= 15):\n", bool_mask_combined)
combined_indexed_data = data[bool_mask_combined]
print("Elements between 5 and 15:\n", combined_indexed_data)

# Example of 'axis' in action (e.g., sum along an axis)
print("\n--- Understanding 'axis' ---")
print("Original 'data' array:\n", data)

# Sum along axis=0 (column-wise sum)
sum_axis_0 = np.sum(data, axis=0)
print("\nSum along axis=0 (sum of columns):\n", sum_axis_0)

# Sum along axis=1 (row-wise sum)
sum_axis_1 = np.sum(data, axis=1)
print("Sum along axis=1 (sum of rows):\n", sum_axis_1)
```

6. Array Concatenation and Stacking

Combining arrays is a common operation.

- `np.concatenate((array1, array2, ...), axis=...)`: Joins a sequence of arrays along an existing axis.
- `np.stack((array1, array2, ...), axis=...)`: Joins a sequence of arrays along a *new* axis.
- `np.c_[]`: A convenient shorthand for column-wise concatenation (stacking 1D arrays as columns or joining 2D arrays column-wise).
- `np.r_[]`: A convenient shorthand for row-wise concatenation.

```
print("\n--- Concatenation and Stacking ---")

arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
arr3 = np.array([[7, 8, 9]]) # Must be 2D for column-wise concatenation with 2D arrays

# Concatenate 1D arrays
concat_1d = np.concatenate((arr1, arr2))
print("Concatenate 1D arrays (arr1, arr2):\n", concat_1d)

# Create 2D arrays for further examples
mat1 = np.array([[10, 11], [12, 13]])
mat2 = np.array([[14, 15], [16, 17]])

print("\nMatrix 1:\n", mat1)
print("Matrix 2:\n", mat2)

# Concatenate along axis 0 (rows)
concat_axis_0 = np.concatenate((mat1, mat2), axis=0)
print("\nConcatenate along axis 0 (rows):\n", concat_axis_0)

# Concatenate along axis 1 (columns)
concat_axis_1 = np.concatenate((mat1, mat2), axis=1)
print("\nConcatenate along axis 1 (columns):\n", concat_axis_1)

# Stack along a new axis (creates a 3D array)
stack_axis_0 = np.stack((mat1, mat2), axis=0)
print("\nStack along axis 0 (new 0th dimension):\n", stack_axis_0)
print("Shape of stack_axis_0:", stack_axis_0.shape)

stack_axis_1 = np.stack((mat1, mat2), axis=1)
print("\nStack along axis 1 (new 1st dimension):\n", stack_axis_1)
print("Shape of stack_axis_1:", stack_axis_1.shape)

# np.c_[] for column-wise concatenation
# Original example from prompt:
a_col = np.array([1, 2, 3])
b_col = np.array([4, 5, 6])
result_c_ = np.c_[a_col, b_col] # Makes them columns and stacks
print("\nnp.c_[a_col, b_col]:\n", result_c_)
print("Shape:", result_c_.shape)

# This is equivalent to:
# np.concatenate((a_col[:, np.newaxis], b_col[:, np.newaxis]), axis=1)
# or np.stack((a_col, b_col), axis=1) if a_col and b_col are 1D
```

7. Statistical Operations

NumPy provides powerful functions for common statistical calculations.

- `np.mean(array, axis=...)`: Calculates the arithmetic mean.
- `np.std(array, axis=...)`: Calculates the standard deviation.
- `np.argmax(array, axis=...)`: Returns the indices of the maximum values along an axis.

```
print("\n--- Statistical Operations ---")

# Let's use our 'data' array again: np.arange(1, 26).reshape(5, 5)
print("Original 'data' array:\n", data)

# Mean and Standard Deviation of the entire array
mean_total = np.mean(data)
std_total = np.std(data)
print(f"\nMean of entire array: {mean_total:.2f}")
print(f"Standard deviation of entire array: {std_total:.2f}")

# Mean and Standard Deviation along axis=0 (column-wise)
mean_cols = np.mean(data, axis=0)
std_cols = np.std(data, axis=0)
print("\nMean along axis=0 (column means):\n", mean_cols)
print("Standard deviation along axis=0 (column stds):\n", std_cols)

# Mean and Standard Deviation along axis=1 (row-wise)
mean_rows = np.mean(data, axis=1)
std_rows = np.std(data, axis=1)
print("\nMean along axis=1 (row means):\n", mean_rows)
print("Standard deviation along axis=1 (row stds):\n", std_rows)

# argmax: Find the index of the maximum value
row_with_max = np.array([10, 5, 20, 15, 8])
max_index = np.argmax(row_with_max)
print(f"\nArray: {row_with_max}")
print(f"Index of maximum value: {max_index} (value is {row_with_max[max_index]})")

# argmax along an axis
print("\nArgmax along axis=0 (column-wise max index):", np.argmax(data, axis=0))
print("Argmax along axis=1 (row-wise max index):", np.argmax(data, axis=1))
```

8. Random Number Generation

The `np.random` module is essential for tasks like initializing model weights, shuffling data, or generating synthetic datasets.

- `np.random.randn(d0, d1, ..., dn)`: Returns samples from the "standard normal" distribution (mean 0, variance 1).
- `np.random.normal(loc=0.0, scale=1.0, size=None)`: Returns samples from a normal (Gaussian) distribution with specified `loc` (mean) and `scale` (standard deviation).

```
print("\n--- Random Number Generation ---")

# Generate 5 random numbers from a standard normal distribution (mean=0, std=1)
rand_n_array = np.random.randn(5)
print("np.random.randn(5):\n", rand_n_array)

# Generate a 2x3 array from a standard normal distribution
rand_n_matrix = np.random.randn(2, 3)
print("\nnp.random.randn(2, 3):\n", rand_n_matrix)

# Generate 5 random numbers from a normal distribution with mean=10, std=2
normal_array = np.random.normal(loc=10.0, scale=2.0, size=5)
print("\nnp.random.normal(loc=10.0, scale=2.0, size=5):\n", normal_array)
```

```
# Generate a 3x2 array from a normal distribution with mean=-5, std=1.5
normal_matrix = np.random.normal(loc=-5.0, scale=1.5, size=(3, 2))
print("\nnp.random.normal(loc=-5.0, scale=1.5, size=(3, 2)):\n", normal_matrix)
```

9. Shuffling and Permutation

These functions are crucial for randomizing data, for example, before splitting into training and testing sets.

- `np.random.shuffle(x)`: Modifies the sequence `x` in-place by shuffling its contents. Only shuffles the first dimension for multi-dimensional arrays.
- `np.random.permutation(x)`: Returns a randomly permuted copy of `x`. If `x` is an integer, it returns a permutation of `np.arange(x)`. If `x` is an array, it returns a shuffled copy, leaving the original array untouched.

```
print("\n--- Shuffling and Permutation ---")

# Example with a 1D array
x = np.array([1, 2, 3, 4, 5])
print("Original 1D array 'x':", x)

# shuffle modifies x in-place
np.random.shuffle(x)
print("Array 'x' after np.random.shuffle(x):", x) # x is now shuffled

# permutation returns a new array, leaving x untouched
y = np.random.permutation(x)
print("New array 'y' after np.random.permutation(x):", y) # y is shuffled
print("Array 'x' after np.random.permutation(x) (x remains unchanged):", x) # x stays as is

# Example with a 2D array (shuffles rows)
matrix_to_shuffle = np.arange(1, 10).reshape(3, 3)
print("\nOriginal 2D array 'matrix_to_shuffle':\n", matrix_to_shuffle)

# shuffle on 2D array shuffles rows in-place
np.random.shuffle(matrix_to_shuffle)
print("matrix_to_shuffle (rows shuffled):\n", matrix_to_shuffle)

# permutation on 2D array returns a shuffled copy of rows
permuted_matrix = np.random.permutation(matrix_to_shuffle)
print("permuted_matrix (rows shuffled copy):\n", permuted_matrix)
print("matrix_to_shuffle (original unchanged):\n", matrix_to_shuffle)
```